

A RECONFIGURABLE FUNCTIONAL UNIT FOR AN ADAPTIVE DYNAMIC EXTENSIBLE PROCESSOR

Noori, Hamid

Department of Informatics, Graduate School of Information Science and Electrical Engineering,
Kyushu University

Mehdipour, Farhad

Computer Engineering and Information Technology Department, Amirkabir University of Technology

Murakami, Kazuaki

Department of Informatics, Graduate School of Information Science and Electrical Engineering,
Kyushu University

Inoue, Koji

Department of Informatics, Graduate School of Information Science and Electrical Engineering,
Kyushu University

他

<https://hdl.handle.net/2324/6794494>

出版情報 : Proc. of 2006 International Conference on Field Programmable Logic and Applications
(FPL 2006), pp.781-784, 2006-08. International Conference on Field Programmable Logic and
Applications

バージョン :

権利関係 :

A RECONFIGURABLE FUNCTIONAL UNIT FOR AN ADAPTIVE DYNAMIC EXTENSIBLE PROCESSOR

Hamid Noori^{}, Farhad Mehdipour[†], Kazuaki Murakami^{*}, Koji Inoue^{*} and Morteza SahebZamani[†]*

^{*}Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-1 Kasuga-koen, Kasuga, Fukuoka, Japan
noori@c.csce.kyushu-u.ac.jp,
{murakami,inoue}@i.kyushu-u.ac.jp

[†]Computer Engineering and Information Technology Department, Amirkabir University of Technology, #424 Hafez Ave., Tehran, Iran
{mehdipur,szamani}@aut.ac.ir

ABSTRACT

This paper presents a reconfigurable functional unit (RFU) for an adaptive dynamic extensible processor. The processor can tune its extended instructions to the target applications, after chip-fabrication. The custom instructions (CIs) are generated deploying the hot basic blocks during the training mode. In the normal mode, CIs are executed on the RFU. A quantitative approach was used for designing the RFU. The RFU is a matrix of functional units with 8 inputs and 6 outputs. Performance is enhanced up to 1.25 using the proposed RFU for 22 applications of Mibench. This processor needs no extra opcodes for CIs, new compiler, source code modification and recompilation.

1. INTRODUCTION

One method for providing enhanced performance is application-specific instruction set extension. In this method, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to custom functional units. Instruction set extension improves performance and also maintains a degree of system programmability, which enables them to be utilized with more flexibility. The main problem with this method is that there are significant non-recurring engineering costs associated with their implementation.

In our approach, an Adaptive dynaMic extensiBIE processoR (AMBER) is presented in which the CIs are adapted to the target applications and generated after chip-fabrication, fully transparently and automatically. This approach reduces the design time and cost drastically. Our CIs are generated by exploiting the HBBs. An HBB is a basic block that is executed more than a given threshold. We propose an RFU to support a wide range of generated CIs. Our 8-input, 6-output RFU is a coarse grain accelerator based on a matrix of functional units (FUs). It is tightly coupled with the base processor. In this method, there is no need to add extra opcodes for CIs, develop a new compiler, change the source code and recompile it.

2. RELATED WORK

PRISC[1], Chimaera[2], OneChip[3] and XiRisc[4] are some instances of tightly coupled integration of a GPP with fine-grain programmable hardware and ADRES [5] is a sample system with coarse-grain hardware. All of these designs require a new programming model, a new compiler, new opcodes for new instructions, source code modification or recompilation. In our approach, the user just runs the applications on the base processor, and then, generation of custom instructions and handling their execution are done transparently and automatically.

Adaptive dynamic optimization systems such as Turboscalar [6], rePlay [7], PARROT [8], and Warp Processors [9] select frequently executed regions of the code through dynamic profiling, optimize the selected regions and cache/rewrite the optimized version for future occurrences. The execution of the optimized version is carried on by extra tasks sharing the main processor and/or by extra hardware. To overcome the overhead of dynamic optimization, we have defined two modes for the system.

The similar design to ours has been proposed by Clark [10]. However, we use different methods for profiling and generating, mapping and handling execution of CIs. Our RFU is not integrated like other functional units. It shares the available read/write ports. By applying some modifications in the routing resources and locations of inputs, our RFU can handle more CIs.

3. GENERAL OVERVIEW OF AMBER ARCHITECTURE

AMBER has been designed and developed by integrating three main components to the *base processor*, namely *profiler*, *RFU* and *sequencer*. The base processor is a 4-issue in-order RISC processor that supports MIPS instruction set. Profiling of running applications is done by the *profiler* through monitoring the program counter (PC) [11]. *RFU* is a matrix of functional units (FUs) plus a configuration memory. Each CI updates the PC after its

execution finishes, considering original sequence execution, so that the processor can continue from the correct address.

The *sequencer* mainly determines the microcode execution sequence by selecting between the RFU and the processor functional unit. It has a table in which the start addresses of CIs in the object code are specified. The sequencer monitors the PC and compares it to its table entries. When it detects that a CI is going to be executed, it switches from processor functional unit to the RFU, waits for specified clock cycles and lets the RFU finish the execution of the CI and then again switches to the processor functional unit.

AMBER has two operational modes: training mode and normal mode. In the training mode, applications are run on the base processor and profiled. Then, the start addresses of HBBs are detected. Using these addresses, HBBs are read from the object code. CI generation has been limited to one HBB. When these processes are done, the processor switches to the normal mode.

In the normal mode, using the RFU, its configuration data, sequencer and its table, the CIs are executed on the RFU. For more details on AMBER, refer to [11].

4. RFU ARCHITECTURE: A QUANTITATIVE APPROACH

4.1. Tool Flow for Quantitative Approach

We followed a quantitative approach by applying the flow in Fig. 2 for designing RFU, using 22 applications of Mibench [12]. SimpleScalar [13] was utilized as our simulator. The simulator was modified to generate a trace of taken branches and jumps as an input for the profiler for detecting start addresses of HBBs [11]. For each HBB start address, its corresponding basic block is read from the object code. Then DFG is generated for each HBB and passed to the CI generator tool. The mapping tool receives the optimized CIs and maps them on RFU.

Our CI generator, mapping tool and RFU were developed in two phases. In the first phase, we assumed some primary constraints for both CIs and RFU. CIs were generated and mapped on RFU considering these constraints. We concluded a proper architecture for RFU, by analyzing the feedbacks resulted from mapping. After finalizing the RFU, an integrated temporal partitioning and mapping framework was developed for generating CIs. The details of the framework are out of the scope of this paper.

Primary constraints for generating CIs are: a) supporting only fixed-point instructions excluding *multiply*, *divide* and *load* and b) including at most one *store* and at most one *control* instructions. As the primary constraints for the RFU, a matrix of FUs which can support only fixed-point instructions of the base processor was assumed without any limitations on the number of inputs, outputs (I/O) and FUs.

The output of each FU was supposed to be used by the neighbors in the same row and by all other FUs in the lower level rows.

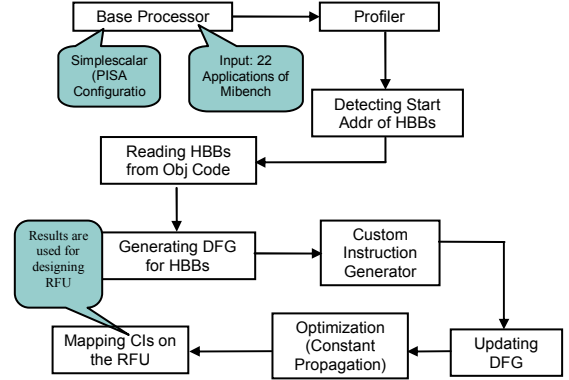


Fig. 1. Tool Flow

Our CI generator receives the DFG of each HBB as an input and then looks for the longest sequence of instructions that can be executed on the RFU. After checking the flow dependence and anti-dependence, more instructions are added to the head and tail of the detected instruction sequence by moving *executable instructions* in the object code. *Executable instructions* are those instructions that can be executed by the RFU. For those parts of the object code where instructions are moved, the object code is rewritten, if these conditions are met.

Mapping is the appropriate positioning of DFG nodes on FUs. Assigning instructions of CI or DFG nodes to FUs is done based on the priority of nodes. The nodes assigned lower value of ASAP (As Soon As Possible) have to be executed earlier. ASAP represents the execution order of nodes according to their dependencies. After calculating ASAP of each node, mapping of nodes is done, starting with lower level nodes to higher level nodes.

4.2. Proposed Architectures for RFU

In this paper, *mapping rate* is defined as the percentage of generated CIs that can be mapped on the RFU for 22 applications of Mibench. We have considered the execution frequency of CIs for measuring the mapping rate as well. All 22 applications were executed till completion. Because execution time varies for each application, for a fair comparison, a weight was assumed for each so that the production of execution time and weight is equal for all.

To determine the proper numbers for RFU inputs and outputs, we mapped our generated CIs on the RFU without considering any constraints. The curves in Fig. 3 show the mapping rate for different numbers of inputs and outputs.

According to the results, eight and six are good candidates for the number of inputs and outputs, respectively. To find the appropriate number for FUs, we similarly measured the mapping rate for various numbers of

FUs. The measurement was done for two cases. Once it was done for CIs that meet our I/O constraints obtained from last experiments, and in the second case, we did not assume any limitation. The two graphs in Fig. 4 show that 16 is a good candidate.

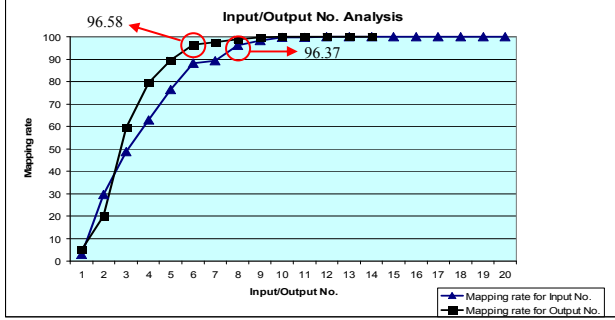


Fig. 2. Mapping rate for different numbers of I/O

We continued similar procedure to specify the width and depth of RFU. Experimental results specify that 6 and 5 are appropriate for width and depth, respectively. By adding the width and depth constraints to previous constraints, the mapping rate will reduce from 94.74% to 93.51%.

Measuring the mapping rate for different numbers of FUs in each row shows that 6, 4, 3, 2 and 1 for first to fifth rows, respectively, are proper candidates. By adding these new constraints, the mapping rate reaches to 92.28%. However, in this architecture, we have assumed that the inputs of the RFU can be accessed by any FU directly and there are direct connections from the output of each row to the input of other lower rows. Moreover, each FU can have inputs (outputs) from (to) the left and right FUs.

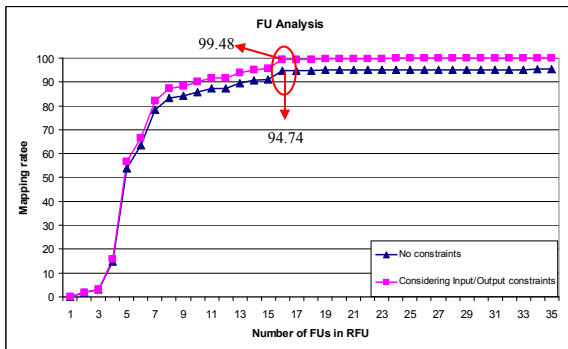


Fig. 3. Mapping rate for different numbers of FUs

To make the architecture more realistic, we assumed that all inputs are applied only to the first row. We also limited the number of connections. The outputs of each row can be used only by all FUs in the subsequent row. All connections to and from neighboring FUs were deleted. In this architecture, for transferring input data to rows below the first row, or transferring the output of one row to the input of FUs in a non-subsequent row, *move* instructions

should be inserted on the intermediate FUs. With these limitations, the mapping rate decreases to 77.53%.

To improve the mapping rate, we examined many different configurations and structures. According to the mapping rate results, we reached to the following architecture depicted in Fig. 5.

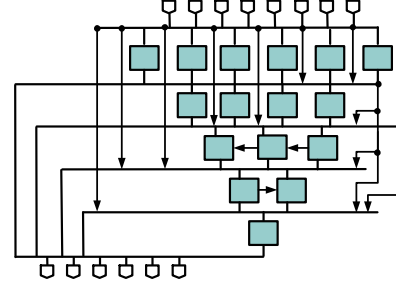


Fig. 4. Optimized RFU architecture

In this architecture, to facilitate data accessing for FUs and reduce the inserted move instructions (which occupy FUs), four other longer connections were added to the base connections. Base connections connect the output of each row to the inputs of subsequent rows. These four longer connections connects row 1 to rows 3, 4 and 5 and row 2 to row 4. We also distributed the input ports among rows. 7, 2, 2, 2, 1 are the number of inputs for the first to fifth rows, respectively that can facilitate access to inputs directly for all rows. The number of inputs for the RFU is 8 and these 14 inputs are generated by replicating the main 8 inputs. In the third and fourth rows, three uni-directional connections to the neighboring FUs, were added to support CIs with critical path longer than 5.

Table 1. Number of required functions in each row

Row No.	Type 1	Type 2	Type 3
1	2	6	4
2	3	3	2
3	1	3	2
4	1	2	1
5	1	1	0

Experiments show that each FU of RFU does not need to support all the operations. We defined three types of operations: logical operations (type 1), add/sub/compare (type 2) and shift operations (type 3). Distribution and the number of operation of each type for each row are given in Table 1. Considering all the constraints for the second proposed architecture, the mapping rate increases to 90.48% which is almost 13% better than that for the first architecture. Each configuration needs 308 bits for control signals and 204 bits for immediate values.

5. INTEGRATING RFU AND BASE PROCESSOR

Fig. 6 depicts how the RFU is connected to the base processor. The I/O ports of the processor functional units

have been shared by the RFU. In a conventional processor, the signals for reading registers are generated by the decode stage. In this design, two signals control reading the register file, one comes from decode stage and the other from configuration bits. Fig. 6 shows four outputs for RFU whereas we had mentioned that RFU had 6 outputs. To support RFU with six outputs without adding write port to the register file, we added two registers to the RFU. When the custom instruction has more than four outputs, extra write values are registered. Four of them are written in one cycle and the remaining ones in the next cycle.

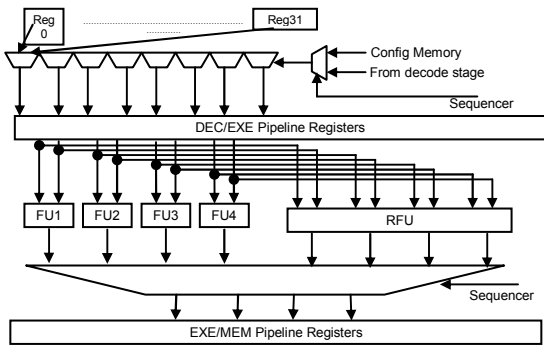


Fig. 5. Integrating RFU with base processor

6. PERFORMANCE EVALUATION

SimpleScalar was used as our simulator framework. As for the base processor, we assumed a 4-issue in-order RISC processor supporting MIPS instruction set with 32KB L1 data cache (1 cycle hit), 32KB L1 instruction cache (1 cycle hit), 1MB unified L2 cache (6 cycle hit), 64 RUU size and 64 fetch queue size. We assumed a variable delay for our RFU which depends on the length of the critical path after mapping a CI on the RFU. For the first row, we supposed that it took one clock cycle and for the other rows 0.5 clock cycle. The delay of RFU for a given CI is calculated as:

$$RFU\ delay = \lceil 1 + (critical_path_length - 1) * 0.5 \rceil$$

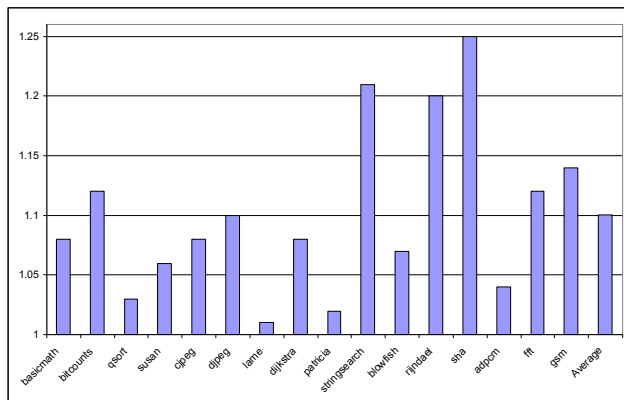


Fig. 6. Speedup obtained for some Mibench App.

7. CONCLUSIONS

Using a quantitative approach, we proposed an RFU for an adaptive dynamic extensible processor. The RFU has 8 inputs and six outputs with 16 FUs. By adding few longer connections between rows and facilitating the input access we could improve the mapping rate by 13%. The performance improvement was up to 25% and the average speedup was 1.10.

8. ACKNOWLEDGEMENTS

This research was supported in part by the Grant-in-Aid for Creative Basic Research, 14GS0218, and for Encouragement of Young Scientists (A), 17680005.

9. REFERENCES

- [1] R. Razdan, and M. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. MICRO-27*, Nov. 1994, pp. 172–180.
- [2] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera reconfigurable functional unit," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, Apr. 1997, pp. 87–96.
- [3] J. E. Carrillo, and P. Chow, "The effect of reconfigurable units in superscalar processors," in *Proc. of the 2001 ACM/SIGDA FPGAs*, 2002, pp. 141–150.
- [4] A. Lodi, et al., "A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1876–1886, 2003.
- [5] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereinsg, "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study," in *Proc. Design, Automation and Test in Europe*, 2004, pp. 1224–1229.
- [6] B. Black, and J. P. Shen, "Turboscalar: A High Frequency High IPC Microarchitecture," in *Proc. ISCA-27*, 2000.
- [7] S. Patel, and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization," *IEEE Transaction on Computers*, vol. 50, no. 6, pp. 590–608, 2001.
- [8] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson, "Power Awareness through Selective Dynamically Optimized Traces," in *Proc. ISCA-31*, 2004, pp. 162–175.
- [9] <http://www.cs.ucr.edu/~vahid/warp/>
- [10] N. Clark, et al., "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *Proc. MICRO-37*, 2004, pp. 30–40.
- [11] H. Noori, et al., "A General Overview of an Adaptive Dynamic Extensible Processor," in *Proc. Workshop on Introspective Architecture*, 2006.
- [12] www.eecs.umich.edu/mibench
- [13] www.SimpleScalar.com